International Journal of Computer Science Engineering
and Information Technology Research (IJCSEITR)
ISSN(P): 2249-6831; ISSN(E): 2249-7943
Vol. 4, Issue 2, Apr 2014, 97-108

TRANS STELLAR
• Journal Publications • Research Consultancy

DESIGN AND DEVELOPMENT OF SOFTWARE INSPECTION TOOL

RAKHEE KUNDU[1] & UMESH KULKARNI[2]

[1]Department of Computer Engineering, Alamuri Ratnamala Institute of Engineering and Technology, Maharashtra, India

[2]Department of Computer Engineering, Vidyalankar Institute of Technology, Maharashtra, India

ABSTRACT

Software inspection is a technique for detecting problems in software early in the lifecycle. Software-inspection process is when the inspectors attempt to find defects by scrutinizing the code in detail. Often, a team will have a checklist of generic and domain-specific rules that must be followed, and the team's task is to find violations of those rules. This paper describes the design and implementation of a tool for helping inspectors navigate this complexity, by providing a means for a user to reason about the deep structure of the code at a high level of detail. This tool, named Code Surfer TM, provides access to and answers queries about—a range of different representations of a program, all created by performing advanced static analysis on the program.

KEYWORDS: Abstract Syntax Tree, Program Dependence Graph (PDG), Predecessor, Slicing, Successor

INTRODUCTION

Software inspection is a technique for detecting problems in software early in the lifecycle. It was introduced by Fagan in 1976 [16] and, since then, it has attracted support as a software engineering best practice. A key phase in the software-inspection process is when the inspectors attempt to find defects by scrutinizing the code in detail. Often, a team will have a checklist of generic and domain-specific rules that must be followed, and the team's task is to find violations of those rules. For example, in a checklist used at NASA for programs written in C [32], one generic rule is "Does code that writes to dynamically allocated memory via a pointer first check for a valid (nonzero) pointer?"Unfortunately, it can be very difficult to manually find violation of this kind of rule.

It may be easy to find violations for small programs, but, even for moderately sized programs with multiple pointer indirections, the complexity can quickly thwart manual attempts at under-standing. This paper describes the design and implementation of a tool for helping inspectors navigate this complexity, by providing a means for a user to reason about the deep structure of the code at a high level o detail This tool, named Code Surfer TM, provides access to and answers queries about—a range of different representations of a program, all created by performing advanced static analysis on the program. These representations go far beyond those provided by traditional program-browsing tools and include an accurate call graph, the results of whole-program pointer analysis, and the program's system dependence graph.

This project is the design and implementation of a C program inspection tool for helping inspectors navigate this complexity, by providing a means for a user to reason about the deep structure of the code at a high level of detail. This tool aims slicing as a main ingredient for software inspection provides access to and answers queries about—a range of different representations of a program, all created by performing advanced static analysis on the program. These representations go far beyond those provided by traditional program-browsing tools and include the program

dependence graph. The standard queries on the program dependence graph such as predecessors and successors, slicing backward and forward are of much use in program understanding.

This project describes a language-independent program representation—the *program dependence graph* and discusses how program dependence graphs, together with operations such as program slicing, can form the basis for powerful programming tools that address the problems listed above.

REPRESENTATIONAL APPROACH OF THE MODULES

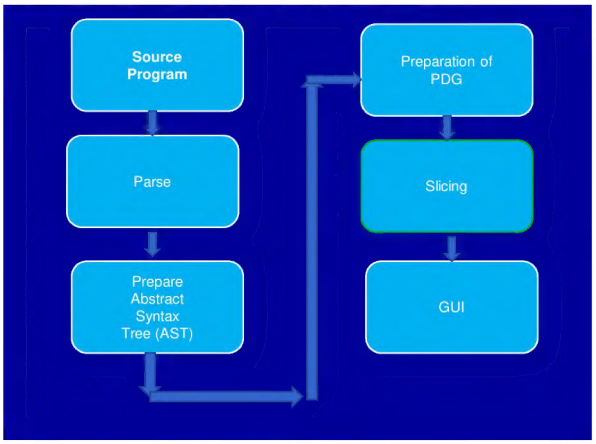

Figure 1

As shown in the figure above a source program which is to be inspected is given as an input using GUI, the program is parsed and abstract syntax tree is constructed. The PDG is generated to understand the main flow of the program and then by slicing the program using forward, backward, predessor or successor approach the code is segmented for further analysis. Then CFG algorithm is applied on it to obtain dominator tree and post dominator tree. The control dependence graph is constructed to understand the dependancing of a particular variable in the entire program and its linkage with other functions and methods. Using this tool it will be easy to find out bugs in the program and their influence on the program control flow will be understood.

PROGRAM DEPENDENCE GRAPH (PDG)

Different definitions of program dependence representations have been given, depending on the intended application; however, they are all variations on a theme and share the common feature of having explicit representations of both control dependences and data dependences. We define *program dependence graphs*, which can be used to represent single procedure programs; that is, programs that consist of a single main procedure, with no procedure or function calls. The *program dependence graphs (or PDG) for a program P, denoted by GP, is a directed graphs whose vertices are* connected by several kinds of edges. The vertices in *GP* represent the assignment statements and predicates of *P*. In addition, *GP* includes a special *Entry* vertex, and also includes one *Initial definition* vertex for every variable *x* that may be used before being defined. (This vertex represents an assignment to the variable from the initial state).

The edges of *GP* represent *control* and *data* dependences. The intuitive meaning of a control dependence edge from vertex *v* to vertex *w* is the following: if the program component represented by vertex *v* is evaluated during program execution and its value matches the label on the edge, then, assuming that the program terminates normally, the component represented by *w* will eventually execute; however, if the value does not match the label on the edge, then the component

represented by *w* may never execute. (By definition, the *Entry* vertex always evaluates to **true**.) For the restricted language under consideration here, control dependence edges reflect the nesting structure of the program (*i.e.*, there is an edge labeled **true** from the vertex that represents a *while* predicate to all vertices that represent statements nested immediately within the loop; there is an edge labeled **true** from the vertex that represents an *if* predicate to all vertices that represent statements nested immediately within the true branch of the *if*, and an edge labeled **false** to all vertices that represent statements nested immediately within the false branch; there is an edge labeled **true** from the *Entry* vertex to all vertices that represent statements that are not nested inside any *while* loop or *if* statement). Data dependence edges include both *flow* dependence edges and use-def dependence edges.3 Flow dependence edges represent possible flow of values, *i.e.*, there is a flow dependence edge from vertex *v* to vertex *w* if vertex *v* represents a program component that assigns a value to some variable *x*, vertex *w* represents a component that uses the value of variable x, and there is a x definition clear path from v to w in the program's control flow graph.

Backward Slicing

A backward slice with respect to a set of starting points S answers the question "What points in the program does S depend on?" The control-dependence edges are used to determine how control could have reached S, and the data dependence edges are used to determine how the variables used at S received their values.

```
int main( ) {
   int sum = 0 ;
   int i = 1 ;
   while ( i < 11 ) {
   sum = sum + i ;
    i = i + 1 ;
    }
     printf("%d\n", sum) ;
   printf("%d\n", i ) ;
   }
```

Backward slice from: printf("%d\n", i) ; is given by the program subset that may affect variable i in underlined printf();

```
int main( ) {
   int sum = 0 ;
   int i = 1 ;
   while ( i < 11 ) {
        sum = sum + i ;
        i = i + 1 ;
```

```
    }
    printf("%d\n", sum) ;
    printf("%d\n",i) ;
}
```

Forward Slicing

A forward slice with respect to a set of starting points S answers the question "What points in the program depend on S?" In this also we make use of control and data dependence edges.

```
int main( ) {
    int sum = 0 ;
    int i = 1 ;
    while ( i < 11 ) {
        sum = sum + i ;
        i = i + 1 ;
    }
    printf("%d\n", sum) ;
    printf("%d\n",i) ;
}
```

Forward slice from: sum = 0 is given by the program subset that may affected by variable sum in *int sum =0* statement

```
int main( ) {
    int sum = 0 ;
    int i = 1 ;
    while ( i < 11 ) {
        sum = sum + i ;
        i = i + 1 ;
    }
    printf("%d\n", sum) ;
    printf("%d\n",i) ;
}
```

Predecessors

It is natural for a user attempting to understand a program to ask "How could variable x have gotten its value

here?" This query can be posed with respect to the control dependences, the data dependences, or both. A program point's data predecessors are the points where the variables used at that point may have gotten their values.

```
1 int main( ) {
2  int sum = 0 ;
3  int i = 1 ;
4 while ( i < 11 ) {
5        sum = sum + i ;
6      i = i + 1 ;
7   }
8  printf("%d\n", sum) ;
9  printf("%d\n",i) ;
10  }
```

The predecessors of variable *sum* at line no 8 is given by

```
1  int main( ) {
2  int sum = 0 ;
3  int i = 1 ;
4  while ( i < 11 ) {
5        sum = sum + i ;
6      i = i + 1;
7   }
8   printf("%d\n", sum) ;
9   printf("%d\n",i) ;
10  }
```

Successors

It is natural for a user attempting to understand a program to ask "Where is the value generated at this point used next?" This query can be posed with respect to the control dependences, the data dependences, or both. A program point's data successors are the points where the variables that were modified at that point are used.

```
1 int main( ) {
2  int sum = 0 ;
3  int i = 1 ;
4 while ( i < 11 ) {
```

```
5       sum = sum + i ;
6      i = i + 1 ;
7   }
8  printf("%d\n", sum) ;
9  printf("%d\n",i) ;
10  }
```

The successors of variable *sum* at line no 2 is given by

```
1  int main( ) {
2  int sum = 0 ;
3  int i = 1 ;
4  while ( i < 11 ) {
5       sum = sum + i ;
6      i = i + 1;
7   }
8   printf("%d\n", sum) ;
            9   printf("%d\n",i) ;
10  }
```

SYSTEM ANALYSIS AND DESIGN

The following figure shows the working nature of the system.

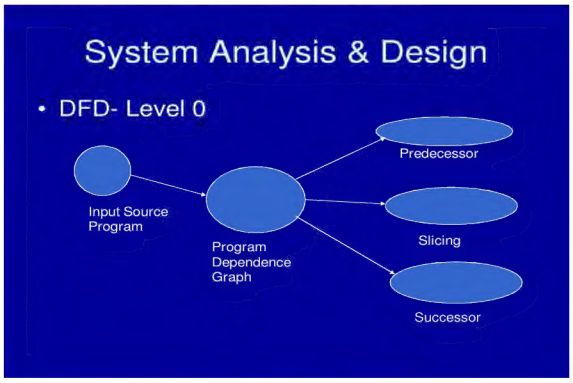

Figure 2

This figure describes the top level view of the system. That is how the system is going to deal with the source code provided to it. First the input is analyzed to produce the intermediate representation in the form of the graph and than subsequent operations is carried out on this representation to produce the result

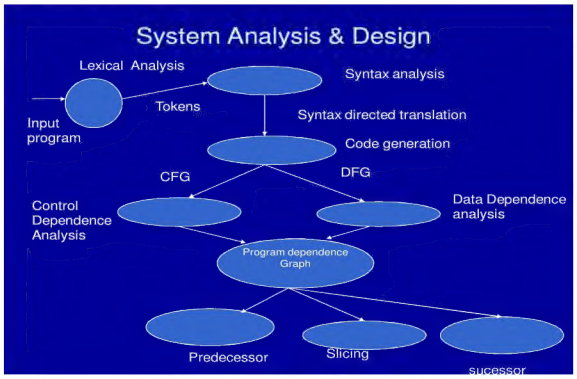

Figure 3

ALGORITHMS AND RELATED THEORY

Computation of Basic Blocks

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

We can construct the basic blocks for a program using algorithm Get Basic Blocks, shown in Figure 1. When we analyze a program's intermediate code for the purpose of performing compiler optimizations, a basic block usually consists of a maximal sequence of intermediate code statements. When we analyze source code, a basic block consists of a maximal sequence of source code statements. We often find it more convenient in the latter case, however, to just treat each source code statement as a basic block.

Algorithm: Get Basic Blocks

Input: A sequence of program statements.

Output: A list of basic blocks with each statement in exactly one basic block.

Method

- Determine the set of *leaders*: the first statements of basic blocks. We use the following rules.
- The first statement in the program is a leader.
- Any statement that is the target of a conditional or an unconditional go to statement is a leader
- Any statement that immediately follows a conditional or an unconditional go to statement is a leader.

Note: control transfer statements such as *while*, *if-else*, *repeat-until*, and *switch* statements are all conditional go to statements".

- Construct the basic blocks using the leaders. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Computing Control Flow Graph

A *control flow graph* (CFG) is a directed graph in which each node represents a basic block and each edge represents the flow of control between basic blocks. To build a CFG we first build basic blocks, and then we add edges that represent control flow between these basic blocks.

After we have constructed basic blocks, we can construct the CFG for a program using algorithm Get CFG, shown in Figure 2. The algorithm also works for the case where each source statement is treated as a basic block. To illustrate, consider Figure 3, which gives the code for program Sums on the left and the CFG for Sums on the right. Node numbers in the CFG correspond to statement numbers in Sums: in the graph, we treat each statement as a basic block. Each node that represents a transfer of control (i.e., 4 and 7) has two labeled edges emanating from it; all other edges are unlabeled. In a CFG, if there is an edge from node *Bi* to node *Bj*, we say that *Bj* is a *successor* of *Bi* and that *Bi* is a *predecessor* of *Bj*. In the example, node 4 has successor nodes 5 and 12, and node 4 has predecessor nodes 3 and 11.

Algorithm: Get CFG

Input: A list of basic blocks for a program where the first block (*B*1) contains the first program statement.

Output: A list of CFG nodes and edges.

Method

- Create *entry* and *exit* nodes; create edge (entry, *B*1); create edges (*Bk*, exit) for each basic block *Bk* that contains an exit from the program.
- Traverse the list of basic blocks and add a CFG edge from each node *Bi* to each node *Bj* if and only if *Bj* can immediately follow *Bi* in some execution sequence, that is, if:
- There is a conditional or unconditional go to statement from the last statement of *Bi* to the first statement of *Bj*, or
- *Bj* immediately follows *Bi* in the order of the program, and *Bi* does not end in an unconditional go to statement.

```
int main( ) {
   int sum = 0 ;
   int i = 1 ;
   while ( i < 11 ) {
        sum = sum + i ;
        i = i + 1 ;
   }
   printf("%d\n", sum) ;
   printf("%d\n",i) ;
}
```

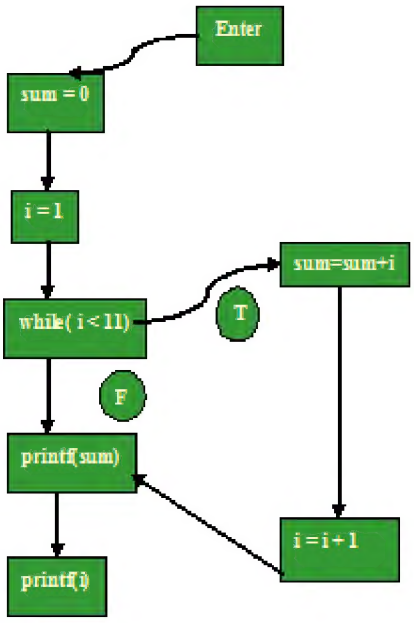

Figure 4

Computing Dominator Tree

A node D in CFG G *dominates* a node W in G if and only if every directed path from entry to W (not including W) contains D. A *dominator tree* is a tree in which the initial node is the entry node, and each node dominates only its descendants in the tree.

Figure 4 gives an algorithm, Compute Dom, for computing dominators for a control flow graph *G*. A key to this algorithm is step 3, where, for each node *n* except the entry node, we initialize the set of dominators to *the set of all nodes in G*. We then iterate through the nodes (except the entry node), and for each node *n*, at step 3, we use the intersection operator to *reduce* the set of nodes listed as dominating *n* to those that actually dominate predecessors of *n*. Thus, we start with an overestimate of the dominators and reduce the sets to get the actual set of dominators

Algorithm: Compute Dom

Input: A control flow graph *G* with set of nodes *N* and initial node *n*0.

Output*: D* (*n*), the set of nodes that dominate *n*, for each node *n* in *G*

Method: Use an iterative approach similar to the data flow analysis algorithm Reaching Defs

1. $D(n0) = \{n0\}$
2. **for** each node *n* in $N - \{n0\}$ **do** $D(n) = N$
3. **while** changes to any $D(n)$ occur **do**
4. **for** *n* in $N-\{n0\}$ **do**
5. $D(n) = \{n\} U (\cap D(p))$ for all immediate predecessors *p* of *n*
6. **endfor**
7. **endwhile**

Sample Control Flow Graph and its Dominator Tree

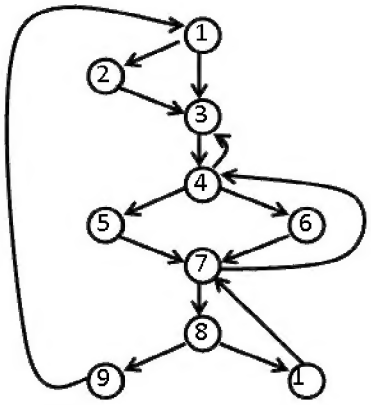

Figure 5

IMPLEMENTATION

The whole system is arranged in the package called **project,** this package contains all the necessary files needed source code and the documentation of the project. The directory contains the two more directories one contains the GUI related code and other contains the back end source code.

The Application Package

This package contains the source code and the necessary make file to compile the source code to produce executable of the GUI program. The code is produced with the help of the QT Designer in C++. The application is the gui which offers all the features of the general purpose text editors. It contains the menu bar which has the File, Help and Tools as the main menus. The New, Open, Save, Save As are the common drop down menus, and in Tools the Slice, Predecessors, Successors, Formatted C code are the drop down menus. The menus are implemented as the components provided by QT designer. The dialog boxes for taking the input from the user are provided using the components provided by the QT designer and C++

The input information collected from the user is outputted in the file called "input.txt" which analyses the C program. Depending on the choice of the user selected the GUI program and the back end which analyses the C program. Depending on the choice of the user selected the GUI program invokes the back end program with appropriate arguments. The back and then performs the appropriate operations depending on the arguments supplied to it and writes the result back to the "output.txt" which is then read and displayed by the GUI program.

CONCLUSIONS

As the dissertation is going on and first two modules are understood and being implemented. In the next part slicing and, program control flow graphs, use of some std. packages like QT Designer, and use of c ++ STL (Std. Template Library) will be studied and implemented for code optimization.

REFERENCES

1. **Dynamic Program Slicing in Understanding of Program Execution by** Bogdan Korel, Jurgen Rilling Department of Computer Science ,Illinois Institute of Technology Chicago, IL 6061 6, **USA** korel@ charlie.iit.edu

2. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis Paolo Tonella IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 29, NO 6, JUNE 2003
3. P. Bishop, R. Bloomfield, S. Guerra, and T. Clement, "Software Criticality Analysis of COTS/SOUP," Proc. Safe comp 2002, Sept. 2002.
4. M. Burke and R. Cytron, "Inter procedural Dependence Analysis and Parallelization," Proc. SIGPLAN '86 Symp Compiler Construction, pp. 162-175, 1986
5. Bell Canada, http://www.iro.umontreal.ca/labs/gelo/datrix, 2001.
6. Language Design and Implementation, pp. 57-66, June 1988. J.R. Cordy, C.D. Halpern, and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects," Computer Languages, vol. 16, no. 1, pp. 97-107, Jan. 1991.
7. ***Design and Implementation of a Fine-Grained Software Inspection Tool*** Paul Anderson, Thomas Reps, and Tim Teitelbaum IEEE TRANSACTIONS on SOFTWARE ENGINEERING, VOL. 29, NO 8, AUGUST 2003 721